Revisiting Sorting for GPGPU Stream Architectures[1]

Duane Merrill (dgm4d@virginia.edu) Andrew Grimshaw (grimshaw@virginia.edu)

Abstract. This paper presents efficient strategies for sorting large sequences of fixed-length keys (and values) using GPGPU stream processors. Compared to the state-of-the-art, our radix sorting methods exhibit speedup of at least 2x for all generations of NVIDIA GPGPUs, and up to 3.7x for current GT200-based models. Our implementations demonstrate sorting rates of 482 million key-value pairs per second, and 550 million keys per second (32-bit). For this domain of sorting problems, we believe our sorting primitive to be the fastest available for any fully-programmable microarchitecture.

These results motivate a different breed of parallel primitives for GPGPU stream architectures that can better exploit the memory and computational resources while maintaining the flexibility of a reusable component. Our sorting performance is derived from a parallel scan stream primitive that has been generalized in two ways: (1) with local interfaces for producer/consumer operations (*visiting logic*), and (2) with interfaces for performing multiple related, concurrent prefix scans (*multi-scan*).

As part of this work, we demonstrate a method for encoding multiple compaction problems into a single, composite parallel scan. This technique yields a 2.5x speedup over bitonic sorting networks for small problem instances, i.e., sequences that can be entirely sorted within the shared memory local to a single GPU core.

1 Introduction

The transformation of the fixed-function graphics processing unit into a fully-programmable, high-bandwidth coprocessor (GPGPU) has yielded a wealth of data-parallel performance opportunities. As a new and disruptive genre of microarchitecture, it is important to establish efficient computational primitives for the corresponding programming model. Software primitives promote software flexibility via abstraction and reuse, and much effort has been spent investigating efficient primitives for GPGPU stream architectures [1].

As a reusable primitive, performance concerns of speed and efficiency are top priorities for parallel sorting routines. Sorting techniques that involve partitioning or merging strategies are particularly amenable for GPGPU architectures: they are highly parallelizable and the computational granularity of concurrent tasks is miniscule. This paper is concerned with the problem of sorting large sequences of elements, specifically sequences comprised of hundreds-of-thousands or millions of fixed-length, numerical keys. We consider two varieties of this problem: sequences comprised (a) 32-bit keys paired with 32-bit satellite values; and (b) 32-bit keys only. Our solution strategies, however, can be generalized for keys and values of other sizes.

The need to rank and order data is pervasive. As an algorithmic primitive, sorting facilitates many problems including binary search, finding the closest pair, determining element uniqueness, finding the k^{th} largest element, and identifying outliers [2,3]. Sorting routines are germane to many GPU rendering applications, including shadow and transparency modeling [4], Reyes rendering [5], volume rendering via ray-casting [6], particle rendering and animation [7,8], ray tracing [9], and texture compression [10]. Sorting serves as a procedural step within the construction of KD-trees [11] and bounding volume hierarchies [12,13], both of which are useful data structures for ray tracing, collision detection, visibility culling, photon mapping, point cloud modeling, particle-based fluid simulation, etc. GPGPU sorting has also found use within parallel hashing [14], database acceleration [15,16], data mining [17], and game engine AI [18].

---

[1] This is a condensed version of Technical Report CS2010-03, Department of Computer Science, University of Virginia. February 2010.

1.1 Contributions

We present the design of our strategy for radix sorting on GPGPU stream architectures, demonstrating that our approach is significantly faster than previously published techniques for sorting large sequences of fixed-size numerical keys. We consider the GPGPU sorting algorithms described by Satish et al. [19] (and implemented in the CUDPP data parallel primitives library [20]) to be representative of the current state-of-the-art. Our work demonstrates factors of speedup of at least 2x for all fully-programmable generations of NVIDIA GPUs, and up to 3.7x for current generation models. In addition, our local sorting strategies exhibit up to 2.5x speedup over bitonic networks for small problem instances that can be entirely sorted within the shared memory of a single GPU core (e.g., 128-512 elements).

Revisiting previous sorting comparisons in the literature between many-core CPU and GPU architectures [21], our speedups show older NVIDIA G80-based GPUs to outperform Intel Core2 quad-core processors. We also demonstrate the NVIDIA GT200-based GPUs to outperform cycle-accurate sorting simulations of the unreleased Intel 32-core Larrabee platform by as much as 42%. The Larrabee architecture provides write-coherent caches and alternative styles of synchronization in an effort to provider higher performance for cooperative problems such as sorting [22].

We refer to our approach as a *strategy* [23] because it is a flexible hybrid composition of several different algorithms. We use different algorithms for composing different phases of computation, where each phase is intended to exploit a different memory space or aspect of computation. Because the number of steps needed for each phase is adjustable, our solution is flexible in terms of support for different SIMD widths, shared memory sizes, and can be mated to optimal patterns of device memory transactions.

GPGPU applications strive to maximally utilize both computational and I/O resources; inefficient usage or underutilization of either is often indicative of suboptimal problem decomposition. Our speedup over the CUDPP implementation is due to our improved usage of both: we require 38% fewer bytes to be moved through the global memory subsystem, and a 64% reduction in the number of thread-cycles needed for computation.

There are two contributing factors that have enabled our efficient use of the hardware. The first is our prior work on the development of efficient, memory-bound parallel *prefix scan* routines [24]. Prefix scan is a useful primitive for parallel shared-memory architectures: it allows processing elements to dynamically and cooperatively determine the appropriate memory location(s) into which their output data can be placed. The radix sorting method is a perfect example: keys can be processed in a data-parallel fashion for a given digit-place, but cooperation is needed among processing elements so that each may determine the appropriate destinations for relocating its keys.

The second factor is that we apply an alternative pattern of program composition. Our key insight is that performance depends not only upon the *implementation* of a given primitive, but its *usage* as well. In typical design, stream primitives are invoked by the host program as black-box subroutines. The stream kernel (or short sequence of kernels) facilitates a natural abstraction boundary: kernel steps in a stream are often functionally orthogonal, exhibiting a low degree of coupling with other routines. However, we demonstrate that greater overall system utilization can be achieved by applying a *visitor* pattern [23] of task composition when reusable primitives are extremely memory-bound. The result is a form of semi-explicit *kernel fusion*: orthogonal steps are coalesced using an additional pair of abstraction interfaces in which the primitive invokes (i.e., "visits") application-specific logic for (a) input-generation and (b) post-processing behavior. For example, our radix sorting design combines the prefix scan primitive with *binning* and *scatter* routines: binning decodes the particular numeral represented within a given key and digit place, and scatter relocates keys (and values) based upon the ordering results computed by scan.

This pattern provides an elegant mechanism for increasing the arithmetic intensity of memory-bound reduction and scan primitives. The overall number of memory transactions needed by the application is dramatically reduced because we obviate the need to move intermediate state (e.g., the input/output sequences for scan) through global device memory. The

elimination of load/store instructions also increases the computational efficiency, further allowing our pattern to exploit those resources.

1.2 Organization of the Report

Section 2 presents a brief, relevant overview of GPGPU stream architectures. Section 3 reviews the radix sorting method. Section 4 describes the design of our strategy, providing an overview of our parallel scan primitive and the details for how we extend it to provide stable digit-sorting functionality. Section 5 presents our performance evaluations and Section 6 concludes with a discussion of how our work fits into the landscape of GPGPU sorting methods.

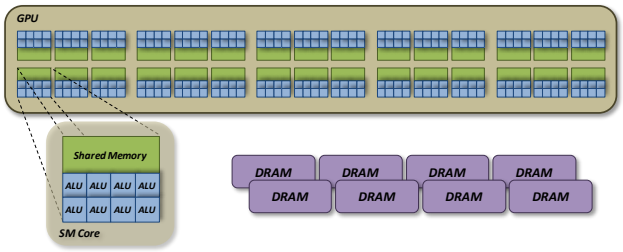

Figure 1. A typical GPGPU organization comprised of 30 SM cores, each having 8 SIMD ALUs and a local shared memory space. Globally-visible DRAM memory is off-chip.

2 Parallel Computation on the GPGPU

2.1 Stream Machine and Programming Models

The GPGPU is capable of efficiently executing large quantities of concurrent, ultra-fine-grained tasks. It is often classified as SPMD (single program, multiple data) in that many hardware-scheduled execution contexts, or *threads*, run copies of the same imperative program, or *kernel*.

The typical GPU processor organization entails a collection of cores (*stream multiprocessors*, or SMs), each of which is comprised of homogeneous processing elements (i.e., ALUs). These SM cores employ local SIMD (single instruction, multiple data) techniques in which a single instruction stream is executed by a fixed-size grouping of threads called a *warp*. Similar to symmetric multithreading (SMT) techniques, each SM contains only enough ALUs to actively execute one or two warps, yet maintains and schedules amongst the execution contexts of many warps. This translates into tens of warp contexts per core, and tens-of-thousands of thread contexts per GPU processor.

Language-level constructs for thread-grouping are provided to facilitate logical problem decomposition in a manner that is also convenient for mapping blocks of threads onto physical SM cores. A two-level grouping hierarchy is often used for programming a single device: a CTA (*cooperative thread array*) of individual threads that share a memory space local to an SM core, and a *grid* of homogeneous CTAs that encapsulates all of the threads for a given kernel.

Threads must explicitly move data from one memory space to another. Cooperation is based on the *bulk-synchronous* model [25]: memory coherence is achieved through the programmatic use of synchronization barriers. Different barriers exist for different spaces: CTA synchronization instructions exist for the coherence of local shared memory, and off-chip global memory is made consistent at the boundaries between serial kernel invocations. An important consequence is that cooperation amongst SM cores requires the invocation of multiple kernel instances. The host orchestrates a *stream* of global data flow by repeatedly invoking new kernel instances, each of which is initially presented with a consistent view of the results from the previous.

2.2 Resource Utilization

It is often less desirable for a given problem to be memory-bound (or, more generally, I/O-bound) than compute-bound. Historically, the trend of growing disparity between processor throughput and I/O bandwidth for successive microprocessor generations has meant that I/O-bound implementations would benefit substantially less from the simple passage of time.

There are two approaches for rebalancing I/O-bound workloads in an effort to obtain better overall system utilization, i.e., for improving *arithmetic intensity*. The first is to increase the amount of local computation in such a way that fewer intermediate results need to be communicated off-chip. Many algorithmic strategies are capable of a variable granularity of computation,

i.e., they can trade redundant computation for fewer I/O operations. For example, increasing the number of bits per radix-digit decreases the total number of digit places that need iterating over. Granularity adjustment has two challenges: (1) the operational details are very application-specific; and (2) linear decreases in I/O often require super-linear increases in dynamic instruction counts and local storage requirements. In general, we see the latter reflected in the "power law of cache misses" [26], which suggests that the influence of cache size (e.g., local storage) upon miss rate (e.g., off-chip memory accesses) follows a power curve in which squaring the cache size typically results in halving the cache misses.

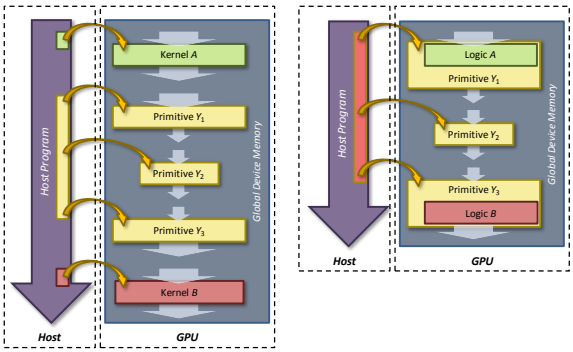

Figure 2. Coalescing separate producer and consumer logic (left) into the kernels of a stream primitive (right).

The second approach is to co-locate sequential steps in the stream pipeline within a single kernel, as illustrated in Figure 2. When data-parallel steps can be combined, the intermediate results can be passed from one step to the next in local registers or shared memory instead of through global, off-chip memory. Data-producing steps are often data-parallel and can be relocated inside the primitive's first kernel, replacing that kernel's original gather operation. The same can be done for a consumer step, i.e., it can replace the scatter logic within the scan primitive's last kernel. Many cooperative primitives have a global cooperation requirement, requiring multiple kernel invocations by the host. As such, the host must actually drive the primitive's kernels which can then in turn "visit" the pre/post steps.

The over-threaded nature of the GPGPU architecture provides predictable and usable "bubbles of opportunity" for extra computation within I/O-bound primitives. If the computational overhead of the visiting logic can fit within the envelope of this bubble, this work can be performed at essentially zero-cost. If the resulting kernel is still memory-bound, we can ratchet up the application-specific arithmetic intensity, if any. This technique is particularly effective for improving the overall system utilization of streams comprised of alternating memory- and compute-bound kernels.

2.3 Analysis and Modeling

One effect of GPGPU latency-hiding is that device saturation occurs when the number of schedulable thread contexts is much greater than the number of ALUs on the GPU die. This causes the system to appear to scale as if it has many more ALUs than it does, i.e., an "effective" processor count for saturated conditions. Unfortunately it can be problematic to model performance using effective processor count and task durations, particularly in terms of abstractions afforded by the programming model (e.g., threads, warps, CTAs, etc.).

As an alternative, we can model performance using the aggregate device throughputs for computation and memory. When saturated, GPU cores typically exist in either a compute-bound or a memory-bound state for the duration of a stream kernel: over-threading is effective at averaging out the particular phases of behavior any given thread is experiencing. Compute-bound kernels proceed at the device's aggregate number of thread-cycles per second (δ_{compute}), and memory-bound kernels proceed at the bandwidth of the memory subsystem (δ_{mem}). By modeling the aggregate computational and memory workloads in terms of cycles and bytes, we can use these throughputs to obtain the corresponding time overheads. Because these GPU workloads are effectively divorced, the overall time overhead will be the larger of the two. We illustrate this with a model of reduction, a subcomponent of our scan primitive.

GPGPUs have a much higher ratio of global communication cost to processor cores than traditional massively-parallel systems. The relative cost of moving data through global memory makes it undesirable for tree-based cooperation between SM cores to be comprised of more than two levels. Consider parallel reduction over a very large network of scalar processors in which the set of processors consume $O(n/p)$ time to generate per-processor sums in parallel, and then $O(\log_2 p)$ time to reduce them in a binary computation tree [27]. For some device-specific task-duration constants c_1 and c_2, we would model saturated runtime as:

$$T_{\text{reduce-scalar}}(n,p) = \frac{c_1 n}{p} + c_2 \log_2 p$$

Our GPGPU version is only comprised of two kernels: (1) a saturating bottom-level kernel comprised of *C* CTAs that each reduces *n*/*C* elements; and (2) an unsaturated top-level kernel in which a single CTA reduces the *C* intermediate results [24]. For inputs at scale, we can dismiss the overhead of the insignificant top-level kernel and drop the $\log_2 p$ component from the model above. Assuming conversion constants c_3 to model reduction operations in terms of cycles and c_4 to model words in terms of bytes, we therefore model saturated GPGPU runtime as follows:

$$T_{\text{reduce-gpu}}(n, \delta_{\text{compute}}, \delta_{\text{mem}}) = max\left(\frac{c_3 n}{\delta_{\text{compute}}}, \frac{c_4 (n + C)}{\delta_{\text{mem}}}\right)$$

3 Parallel Radix Sorting

The radix sorting method relies upon a positional representation for keys, i.e., each key is comprised of an ordered sequence of symbols (i.e., *digits*) specified from least-significant to most-significant. For a specific total ordering of the symbolic alphabet and a given input sequence of keys, the radix sorting method produces a lexicographic ordering of those keys. The process works by iterating over the digit-places from least-significant to most-significant. For each digit-place, the method performs a stable *distribution sort* of the keys based upon their digit at that digit-place. Given an *n*-element sequence of *k*-bit keys and a radix $r = 2^d$, a radix sort of these keys will require *k*/*d* passes of a distribution sort over all *n* keys. The asymptotic work complexity of the distribution sort is *O*(*n*) because each input item needs comparing with only a fixed number of radix digits. With a fixed number of digit-places, the entire sorting process is also *O*(*n*).

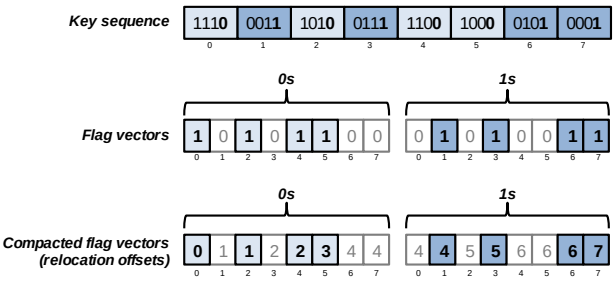

Figure 3. Using scan to perform a radix $r = 2$ distribution sort on the first digit-place of an input sequence.

The distribution sort is the fundamental component of the radix sorting method. In a data-parallel, shared-memory decomposition, each logical processor gathers its key, decodes the specific digit at the given digit-place, and then must cooperate with other processors to determine where the key should be relocated. The relocation offset will be the key's global rank, i.e., the number of keys with "lower" digits at that digit place plus the number of keys having the same digit, yet occurring earlier in the sequence.

The ranking process can be constructed from one or more parallel prefix scans. In its simplest form, this can be done using a binary *split* primitive [28] comprised of two prefix scans over two *n*-element binary flag vectors: the first initialized with 1s for keys whose digit was 0, the second to 1s for keys whose digit was 1. The two scan operations are dependent: the scan of the 1s vector can be seeded with the number of zeros from the 0s scan.[2] After the scans, the i^{th} element in the appropriate compacted flag vector will indicate the relocation offset for the i^{th} key. Alternatively, the two vectors can be concatenated and processed by one large scan as shown in Figure 3.

A simple, naïve GPGPU distribution sort implementation could be constructed from a black-box parallel scan primitive sandwiched between separate *binning* and *scatter* kernels. The binning kernel would be used to create a concatenated flag vector of *rn* elements in global memory. After scanning, the scatter kernel would redistribute the keys (and values) according to the compaction results. This approach suffers from an excessive *O*(*rn*) memory workload that will set a lower bound on the achievable performance: it has a linear coefficient that is exponential in terms of the number of radix digit bits *d*. As a

[2] An optimization for radix $r = 2$ is to obviate the 1s scan: the destination for a 1s key can be determined by adding the total number of 0 keys to the processor-rank and then subtracting the result from compacted 0s vector [30].

Step	Kernel	Purpose	Read Workload	Write Workload
1	*local digit-sort*	Maximize coherence	*n* keys (+ *n* values)	*n* keys (+ *n* values)
2	*Histogram*	Create histograms	*n* keys	*nr/b* counts
3	*bottom-level reduce*	Scan histograms (scan primitive)	*nr/b* counts	(*insignificant constant*)
4	*top-level scan*		(*insignificant constant*)	(*insignificant constant*)
5	*bottom-level scan*		*nr/b* counts + (*insignificant constant*)	*nr/b* offsets
6	*Scatter*	Distribute keys	*nr/b* offsets + *n* keys (+ *n* values)	*n* keys (+ *n* values)

Total Memory Workload: $(k/d)(n)(5r/b + 7)$ keys only
$(k/d)(n)(5r/b + 9)$ with values

Figure 4. GPGPU stream representative of the Satish et al. method for distribution sorting with *d*-bit radix digits, radix $r = 2^d$, local block size of *b* keys, and an *n*-element input sequence of *k*-bit keys.

consequence of this dependency, the overall memory workload for k/d passes would be minimized when the number of radix digit bits $d = 1$, providing little flexibility for adjusting the granularity of computation.

As an alternative, many parallel implementations use a histogram-based strategy [29,30]. For saturating problems, the number of parallel processors is typically smaller than the input sequence size, making it natural to distribute the sequence amongst processors in blocks of *b* keys. Using local resources, each processor can compute an *r*-element histogram of digit-counts. By only sharing these histograms, the global storage requirements are reduced by a factor of *b*. A parallel scan of these histograms provides each processor with the base digit-offsets for its block of keys. These offsets can then be applied to the local key rankings within the block to distribute the keys.

Prior GPGPU radix sort implementations use this approach, treating each CTA as a logical processor operating over a block of *b* keys [31,32,19]. Of these, we consider the radix sort implementation described by Satish et al. to be representative of the current state-of-the-art. Their procedure is depicted in Figure 4. Although the overall memory workload still has a linear coefficient that is exponential in terms of the number of radix digit bits *d*, it is significantly reduced by common block-sizes *b* of 128-1024 keys. Because the number of logical processors grows with problem size, the block-size factor elicits a bathtub effect in which there exists an optimal *d* to produce a minimal memory overhead for a given block size. (E.g., a radix digit size $d = 8$ would provide the minimal memory overhead for their block size $b = 512$ when sorting 32-bit keys and values.) As a point of comparison with our approach, they use $d = 4$ bits, requiring the memory subsystem to process $73.3n$ words for an entire sort of 32-bit keys and values.

We note that their design incorporates a kernel that locally sorts individual blocks of keys by their digits at a specific digit-place. This helps to maximize the coherence of the writes to global memory during the scatter phase. Stream processors typically obtain maximum bandwidth by *coalescing* concurrent memory accesses, i.e. the references made by a SIMD warp that fall within a contiguous memory segment can be combined into one memory transaction. Although computationally expensive, this localized sorting creates ordered subsequences of keys can then be contiguously and efficiently scattered to global memory in a subsequent kernel.

4 Efficient Radix Sorting Strategies

The primary design goal of our radix sorting strategy is to reduce the aggregate memory workload. In addition to being a lower-bound for overall performance, the size of the memory workload also greatly contributes to the size of the computational workload. In this section, we describe how we generalize prefix scan to implement a distribution sort with an asymptotically minimal number of intermediate values that must be exchanged through global device memory. More specifically, we do this by adding two capabilities to the stream kernels that comprise the parallel scan primitive: *visiting logic* and *multi-scan*. We use visiting logic to perform binning and scatter tasks, and multi-scan to compute the prefix sums of radix *r* flag vectors in parallel. Both features serve to increase the arithmetic intensity of our memory-bound primitive.

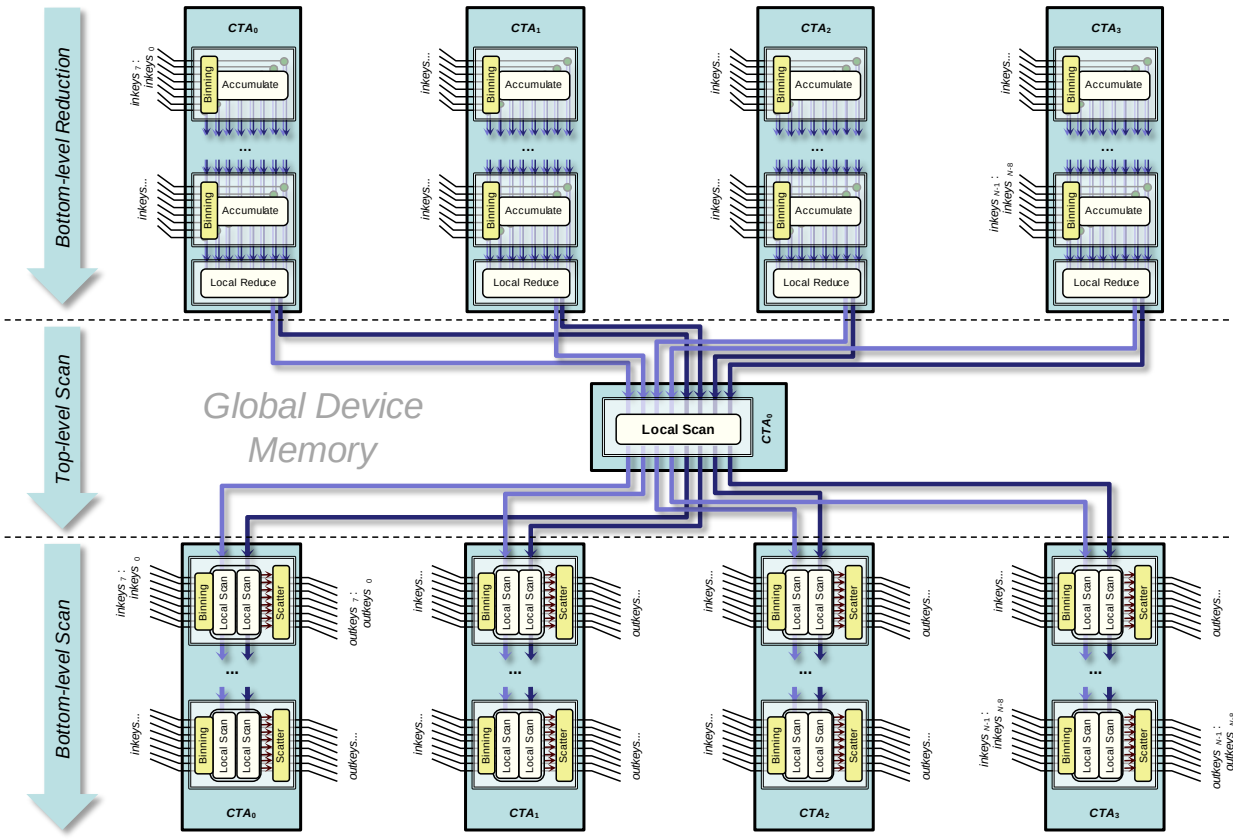

Figure 5. GPGPU stream sequence of kernel invocations for a distribution sort with radix $r = 2$ (two scans). Time flows downwards. This example depicts bottom-level kernel grids comprised of $C = 4$ CTAs, each CTA processing blocks of $b = 8$ values in serial fashion. Data-flows in light-blue indicate live values pertaining to the 0s scan, dark-blue for the 1s scan.

4.1 Meta-strategy

In prior work, we developed several efficient GPGPU scan strategies that use a *two-level reduce-then-scan* meta-strategy for problem decomposition across SM cores [24]. This meta-strategy is composed of three stream kernels: a bottom-level reduction, a top-level scan, and a bottom-level scan. Instead of allocating a unique thread for every input element, our bottom-level kernels dispatch a fixed number C of CTAs in which threads are re-used to process the input sequence in successive blocks of b elements each. Reduction and scan dependencies between blocks are carried in thread-private registers or local shared memory. Figure 5 shows how these kernels have been supplemented with *visiting logic* and *multi-scan* capability.

The bottom-level reduction kernel reduces n inputs into rC partial reductions. Our reduction threads employ a *loop-raking* strategy [33] in which each thread accumulates values in private registers. The standard gather functionality has been replaced with binning logic. When threads in the binning logic read in a block of keys, each decodes the digits at that digit-place for its keys and returns the corresponding digit-counts for each of the r possible digits to the kernel's accumulation logic. After processing their last block, the threads within each CTA perform cooperative reductions in which their accumulated values are reduced into r partial reductions and written out to global device memory, similar to Harris et al. [34].

The single-CTA top-level scan has been generalized to scan a concatenation of rC partial reductions. For radix sorting, a single scan is performed over these sets of partial reductions. The top-level scan is capable of operating in a segmented-scan mode for multi-scan scenarios that produce independent sequences of input.

Step	Kernel	Purpose	Read Workload	Write Workload
1	*bottom-level reduce*	Create flags, compact flags, scatter keys	*n* keys	(*insignificant constant*)
2	*top-level scan*		(*insignificant constant*)	(*insignificant constant*)
3	*bottom-level scan*		*n* keys (+ *n* values) + (*insignificant constant*)	*n* keys (+ *n* values)

Total Memory Workload: (k/d)(3n) keys only
(k/d)(5n) with values

Figure 6. Our distribution sorting GPGPU stream constructed from a parallel multi-scan primitive and visiting binning and scatter kernels with d-bit radix digits, radix $r = 2^d$, and an n-element input sequence of k-bit keys.

In the bottom-level scan kernel, CTAs enact the distribution sort for their portions of the input sequence, seeded with the prefix sums provided by the top-level scan. Each CTA serially reads consecutive blocks of b elements, re-bins them into r local flag vectors, and scans these vectors using a local scan strategy. The scatter logic is then presented with the r prefix sums specific to each key in order to redistribute the keys. It is also responsible for loading and similarly redistributing any satellite values. The aggregate counts for each digit are serially curried into the next b-sized block.

The memory workloads for our distribution-sorting scan primitive are depicted in Figure 6. Only a constant number of memory accesses are used for the storage of intermediate results, and the overall workload no longer has a linear coefficient that is exponential in terms of the number of radix digit bits d. This implies that memory workload will monotonically decrease with increasing d, positioning our strategy to take advantage of any additional computational power that may allow us to increase d in the future. Our strategy can operate with a radix digit size $d \leq 4$ bits on current NVIDIA GPUs before exponentially-growing demands on local storage prevent us from saturating the device. This configuration only requires the memory subsystem to process 40n words for an entire sort of 32-bit keys and values.

4.2 Local Strategy

While the details of the reduction kernel are fairly straightforward, the local operation of our scan kernels warrants some discussion. Of our prior scan primitives, the SRTS variant is the most efficient at processing blocks of contiguous elements [24]. Figure 7 shows how we have augmented the bottom-level SRTS scan kernel with visiting logic to perform binning and scatter tasks, and with multi-scan to compute the local prefix sums of radix r flag vectors in parallel. The figure is illustrated from the point of a single CTA processing a particular block of input values. Threads within the binning logic collectively read b keys, decode them according to the current digit-place, and create the private-register equivalent of r flag vectors of b elements each. The scan logic is replicated r-times, ultimately producing r vectors of b prefix sums each: one for each of the r possible digits.

The scan logic itself is a flexible hierarchy of reduce-then-scan strategies composed of three phases of upsweep/downsweep operation: (1) *thread-independent* processing in registers, shown in blue; (2) *inter-warp cooperation*, shown in orange; and (3) *intra-warp cooperation*, shown in red. The thread-independent phase serves to transition the problem from the block size b into a smaller version that will fit into shared memory and back again. This provides flexibility in terms of facilitating different memory transaction sizes (e.g., 1/2/4-element load/stores) without impacting the size of the shared-memory allocation. In the inter-warp cooperation phase, a single warp serially reduces and scans though the partial reductions placed in shared memory by the other warps in a raking manner similar to [35], transitioning the problem size into one that can be cooperatively processed by a single warp and back again. This provides flexibility in terms of facilitating shared memory allocations of different sizes, supporting alternative SIMD warp sizes, and accommodating arbitrary numbers of warps per CTA. For a given warp-size of w threads, the intra-warp phase implements $\log_2 w$ steps of a Kogge-Stone scan [36] in a synchronization-free SIMD fashion as per [37]. Running totals from the previous block are carried into the SIMD warpscan, incorporated into the prefix sums of the current block's elements, and new running totals are carried out for the next block, all in local shared memory.

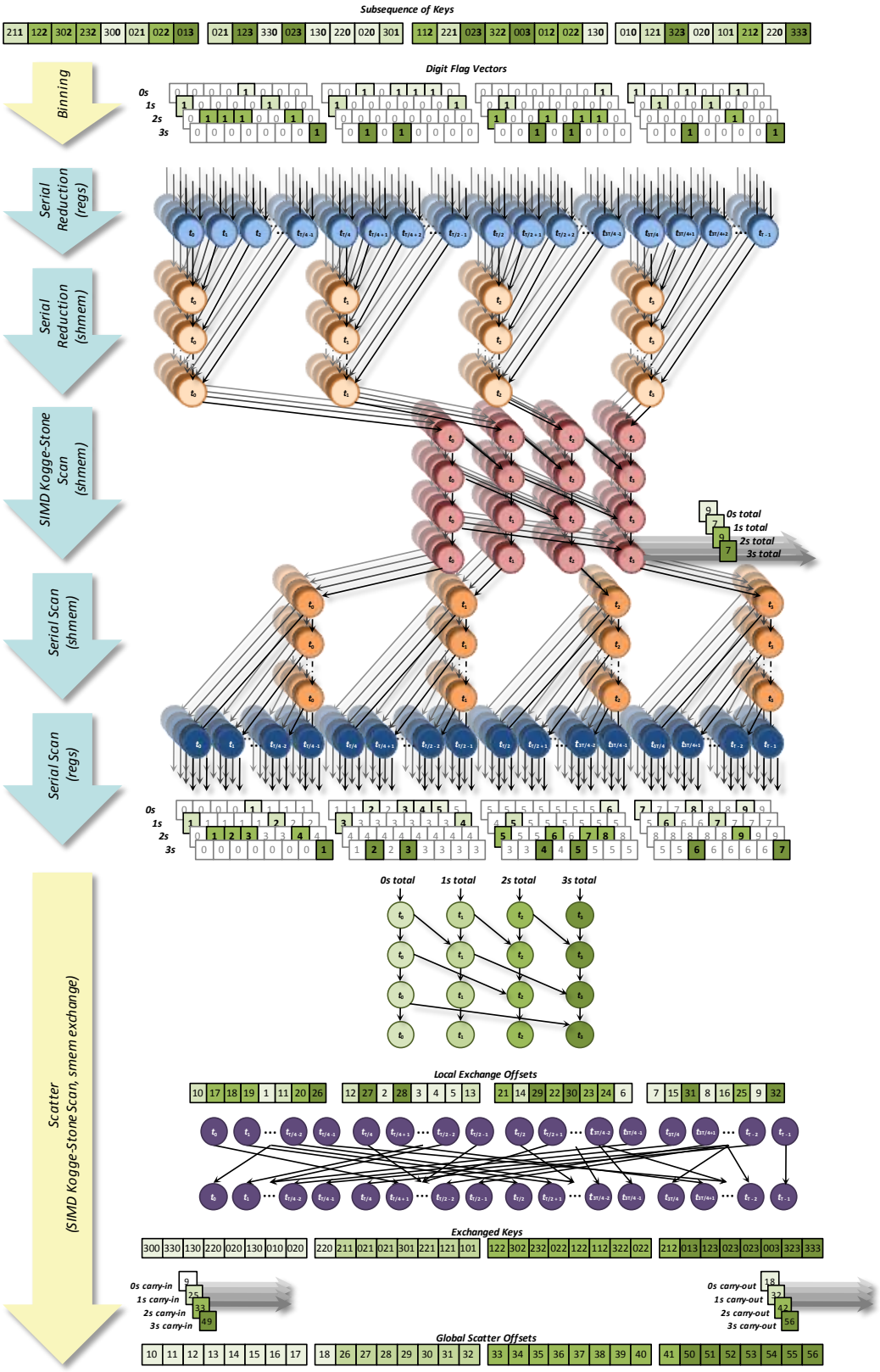

Figure 7. The operation of a generalized SRTS multi-scan CTA that incorporates visiting logic for binning and scatter operations. This figure depicts computation and time flowing downward for an input block size of $b = 32$ keys, a radix $r = 4$ digits, and a warp-size $w = 4$ threads. The five SRTS stages are labeled in light blue, the visiting stages in yellow. Circles indicate the assignment of a given thread t_i to a binary associative task. Flag vector encoding is not shown. The blue thread-independent processing phase is shown to accommodate 2-element (128B) loads/stores.

The scatter operation is provided with the block of *b* keys, the *r* vectors of local prefix sums, the local digit totals, and the incoming running digit totals. Although each scatter thread could use this information to distribute the same keys that it obtained during binning, doing so would result in poor write coherence. Instead we use the local prefix sums to scatter them to local shared memory where consecutive threads can pick up consecutive keys and then scatter them to global device memory with a minimal number of memory transactions. In order to do this, the scatter operation computes the dependencies among prefix sum vectors with a SIMD warpscan of the local digit totals.

Although our two-phase scatter procedure is fairly expensive in terms of dynamic instruction overhead and unavoidable bank conflicts, it is much more efficient than the sorting phase implemented by [19]. Their sorting phase performs *d* iterations of binary-split, exchanging keys (and values) *d* times within shared memory, whereas our approach only exchanges them once.

4.3 Multi-scan Optimization and Performance Modeling

For the purposes of performing a distribution sort, the visiting binning logic returns a set of *compaction* problems (i.e., prefix scans of binary-valued input elements) to the multi-scan component. In this subsection, we describe our method for increasing the efficiency of these compactions. Our binning and scatter operations employ a method of flag vector encoding that takes advantage of the otherwise unused high-order bits of the flag words. By breaking a $b = 512$ element block into two sets of 256-element multi-scans, the scatter logic can encode up to four digit flags within a single 32-bit word. The bit-wise parallelism of 32-bit addition allows us to effectively process four radix digits with a single composite scan. For example, a CTA configured to process a 4-bit digit place can effectively compact all sixteen local digit vectors with only four scan operations.

To evaluate this method, we can decompose the computational overhead of the bottom-level scan kernel in terms of the following tasks: data-movement to/from global memory; digit inspection and encoding of flag vectors in shared memory; shared-memory scanning; decoding local rank from shared memory; and locally exchanging keys and values prior to scatter. For a given key-value pair, each task will incur a fixed cost α in terms of thread-instructions. The flag-encoding and scanning operations will also incur a per-pair cost of β instructions per composite scan. We model the computational workload of the bottom-level scan kernel in terms of thread-instructions as follows:

$$instrs_{\text{scankernel}}(n,r) = n\left(\alpha_{\text{mem}} + \alpha_{\text{encflags}} + \alpha_{\text{scan}} + \alpha_{\text{decflags}} + \alpha_{\text{exchange}} + \frac{r}{4}\left(\beta_{\text{encflags}} + \beta_{\text{scan}}\right)\right)$$

For the NVIDIA GT200 architecture, we have empirically determined $instrs_{\text{scankernel}}(n,r) = n\ (51.4 + r)$. The instruction costs per pair are: $\alpha_{\text{mem}} = 6.3$; $\alpha_{\text{encflags}} = 5.5$; $\alpha_{\text{scan}} = 10.7$; $\alpha_{\text{decflags}} = 13.9$; and $\alpha_{\text{exchange}} = 14.7$. The instruction costs per pair per composite scan are: $\beta_{\text{encflags}} = 2.6$; and $\beta_{\text{scan}} = 1.4$. The result is that our encoding reduces the incremental overhead of *r* to 1.0 instruction per key.

As an alternative, a single-instruction, native implementation of the 32-bit `popc()` intrinsic has been promoted as a mechanism for improving the efficiency of parallel compaction [38]. The `popc()` intrinsic is a bitwise operation that returns the number of bits set in a given word. A local compaction strategy using `popc()` would likely require at least four instructions per element per radix digit: a `ballot()` instruction to cooperatively obtain a 32-bit word with the appropriate bits set, a mask to remove the higher order bits, a `popc()` to extract the prefix sum, and one or more instructions to handle prefix dependencies between multiple `popc()`-words when compacting sequences of more than 32 elements. Because of our lower per-scan costs, our strategy will be more efficient when $r \geq 8$ (two or more composite scans).

5 Evaluation

This section presents the performance of our SRTS-based radix sorting strategy along with the *CudPP* v1.1 implementation as a reference for comparison. Our primary test environment consisted of a Linux platform with an Intel i7 quad-core CPU and an NVIDIA GTX-285 GPU. Our analyses are derived from measurements taken over a suite of ~1,500 randomly-sized input sequences, each initialized with 32-bit keys and values sampled from a uniformly random distribution. Our

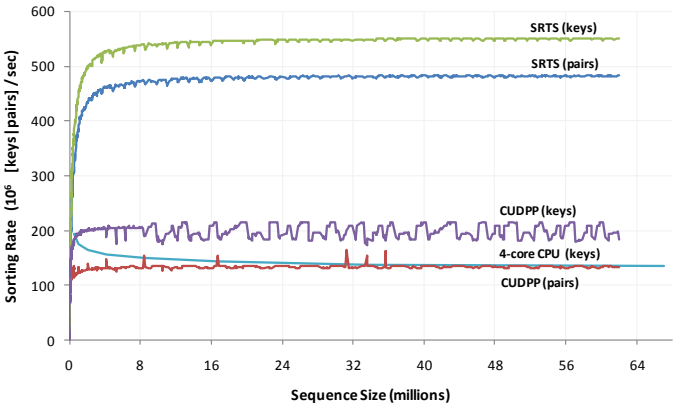

Figure 8. GTX-285 key-value and key-only radix sorting rates for the CUDPP and our 4-bit SRTS-based implementations, overlaid with Chhugani et al. [21] key-only sorting rates for the Intel Core-2 Q9550 quad-core CPU.

Device		*Key-value Rate (10^6 pairs / sec)*		*Keys-only Rate (10^6 keys / sec)*	
Name	*Release Date*	*CUDPP Radix*	***SRTS Radix (speedup)***	*CUDPP Radix*	***SRTS Radix (speedup)***
NVIDIA GTX 285	Q1/2009	134	**482 (3.6x)**	199	**550 (2.8x)**
NVIDIA GTX 280	Q2/2008	117	**428 (3.7x)**	184	**474 (2.6x)**
NVIDIA Tesla C1060	Q2/2008	111	**330 (3.0x)**	176	**471 (2.7x)**
NVIDIA 9800 GTX+	Q3/2008	82	**165 (2.0x)**	111	**226 (2.0x)**
NVIDIA 8800 GT	Q4/2007	63	**129 (2.1x)**	83	**171 (2.1x)**
NVIDIA 9800 GT	Q3/2008	61	**121 (2.0x)**	82	**165 (2.0x)**
NVIDIA 8800 GTX	Q4/2006	57	**116 (2.0x)**	72	**153 (2.1x)**
NVIDIA Quadro FX5600	Q3/2007	55	**110 (2.0x)**	66	**147 (2.2x)**
				Merge [21]	
Intel Q9550 quad-core	Q1/2008			138	
Intel Larrabee 32-core	Cancelled			386	

Figure 9. Saturated sorting rates for input sequences larger than 16M elements.

measurements for elapsed time, dynamic instruction count, warp serializations, memory transactions, etc., are taken directly from GPU hardware performance counters. Our analyses are reflective of *in situ* sorting problems: they preclude the driver overhead and the overheads of staging data to/from the accelerator, allowing us to directly contrast the individual and cumulative performance of the stream kernels involved.

Figure 8 plots the measured radix sorting rates exhibited by our implementation and the CUDPP primitive. For NVIDIA GT200 and G80 GPUs, we best parameterize our strategy with radix r = 16 digits (d = 4 bits). We have also overlaid the keys-only sorting results presented Chhugani et al. for the Intel Core-2 Q9550 quad-core CPU [21], which we believe to be the fastest hand-tuned numerical sorting implementation for multi-core CPUs. As expected, we observe that the radix sorting performances plateau into steady-state as the GPU's resources become saturated. In addition to exhibiting 3.6x and 2.8x speedups over the CUDPP implementation on the same device, our key-value and key-only implementations provide smoother, more consistent performance across the sampled problem sizes.

Recent publications for this genre of sorting problems have set a precedent of comparing the sorting performances of the "best available" many-core GPU and CPU microarchitectures. At the time, Chhugani et al. championed the performances of Intel's fastest consumer-grade Q9550 quad-core processor and cycle-accurate simulations of the 32-core Larrabee platform over G80-based NVIDIA GPUs [21]. Shortly afterward, Satish et al. presented GPGPU performance that was superior to the Q9550 from the newer NVIDIA GT200 architecture [19]. We extended our comparison to a superset of the devices evaluated by these publications. The saturated sorting rates on these devices for input sequences of 16M+ keys are denoted in Figure 9. We observe speedups over Satish et al. (CUDPP) of 3x+ for the GT200 architectures, and 2x+ for the G80 architectures. Using our method, all of the NVIDIA GPUs outperform the Q9550 CPU and, perhaps more strikingly, our sorting rates for GT200 GPUs exhibit up to 1.4x speedup over the previously-dominant cycle-accurate results for 32-core Larrabee.

After leveraging the efficiency of kernel-fusion, we can continue to increase the arithmetic intensity of our radix sorting strategy by increasing the number of radix digit bits d (and thus decreasing the number of distribution sorting passes). Figure 10 shows our average saturated sorting rates for $1 \leq d \leq 5$. We observe that throughput improves as d increases for $d < 5$. When $d \geq 5$, two issues conspire to impair performance, both related to the exponential growth of radix digits $r = 2^d$ that need scanning. The first is that the cumulative computational workload is no longer decreasing with reduced passes. Because the two bottom-level kernels are compute-bound under this load, continuing to increase overall the computational workload will only result in progressively larger slowdowns. The second issue is that increasing local storage requirements (i.e., registers and shared memory) prevent SM saturation: the occupancy per GPU core is reduced from 640 to 256 threads.

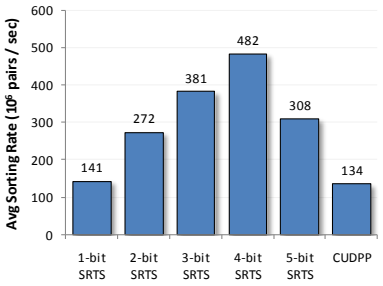

Figure 10. Saturated GTX-285 sorting rates ($n \geq$ 16M key-value pairs).

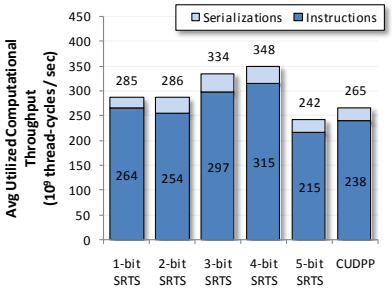

Figure 11. Realized GTX-285 computational throughputs ($n \geq$ 16M key-value pairs).

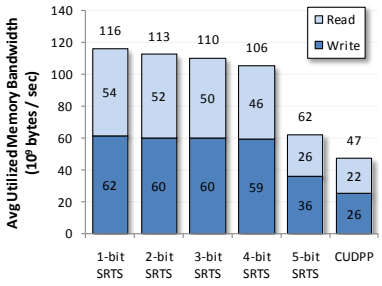

Figure 12. Realized GTX-285 memory bandwidths ($n \geq$ 16M key-value pairs).

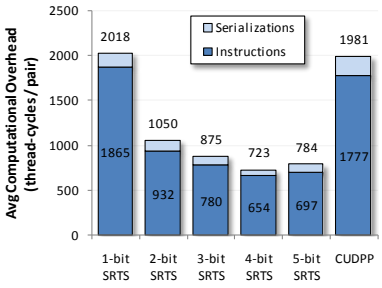

Figure 13. Aggregate GTX-285 computational overhead ($n \geq$ 16M key-value pairs).

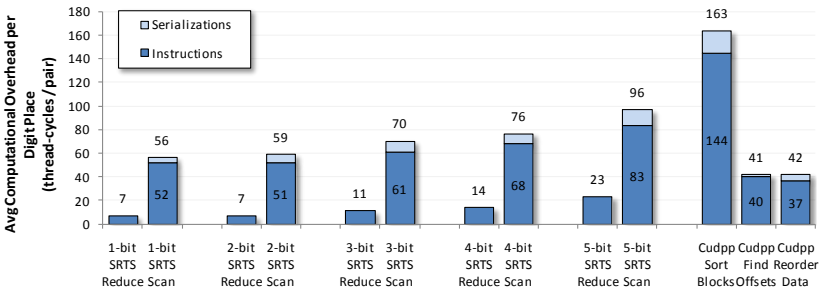

Figure 14. GTX-285 computational overhead per distribution sort ($n \geq$ 16M key-value pairs).

Figures 11 and 12 show the computational throughputs and bandwidths realized by our SRTS variants and the CUDPP implementation. The realistic GTX-285 device maximums are $\delta_{compute} \approx 354$ x10^9 thread-cycles/second and $\delta_{mem} \approx$ 136-149 x10^9 bytes/second. Because their saturating kernels are compute-bound, our 3-bit and 4-bit variants achieve 94+% utilizations of the available computational resources. The 1-bit, 2-bit, and CUDPP implementations have a mixture of compute-bound and memory-bound kernels, resulting in lower overall averages for both. The 5-bit variant illustrates the effects of under-occupied SM cores: its kernels are compute-bound, yet it only utilizes 68% of the available computational throughput.

Our five SRTS columns in Figure 13 illustrate the "bathtub" curve of computational overhead versus digit size: workload decreases with the number of passes over digit-places until the cost of scanning radix digits becomes dominant at $d = 5$. This overhead is inclusive of the number of thread-cycles consumed by scalar instructions as well as the number of stall cycles incurred by the warp-serializations that primarily result from the random exchanges of keys and values in shared memory. The 723 thread-cycles executed per input element by our 4-bit implementation may seem substantial, yet efficiency is 2.7x that of the CUDPP implementation.

Figure 14 presents the computational overheads of the individual kernel invocations that comprise a single digit-place iteration, i.e., distribution sort. For $d > 2$, we observe that the workload deltas between scan kernels double as d is incremented, scaling with r as expected and validating our model of instruction overhead from Section 4.2. Our 1-bit and 2-bit variants don't follow this parameterized model: the optimizing compiler produces different code for them because flag vector encoding yields only one composite scan.

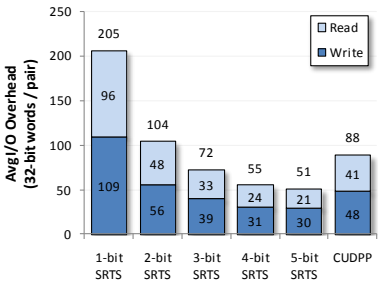

Figure 15. Aggregate GTX-285 memory overhead ($n \geq$ 16M key-value pairs).

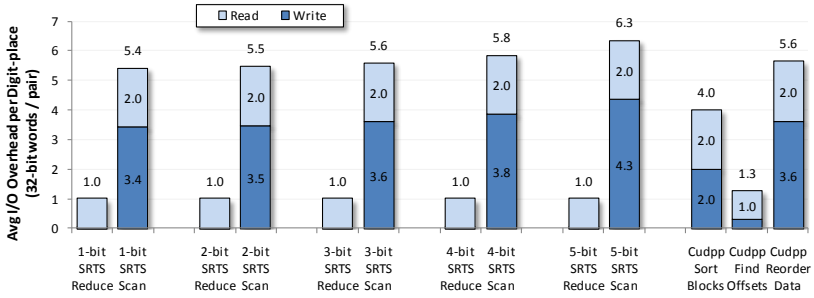

Figure 16. GTX-285 memory overhead per distribution sort ($n \geq$ 16M key-value pairs).

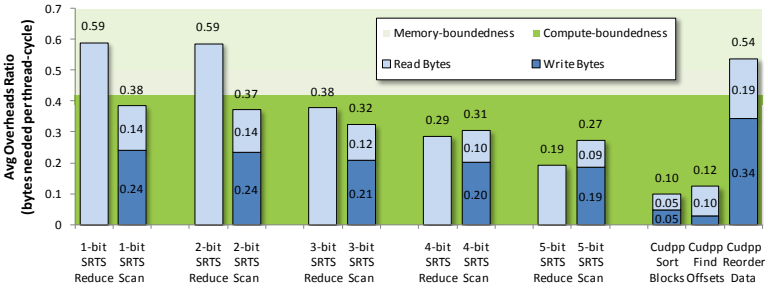

Figure 17. Memory-to-compute workload ratios for individual stream kernels, with the GTX-285 memory wall as a backdrop.

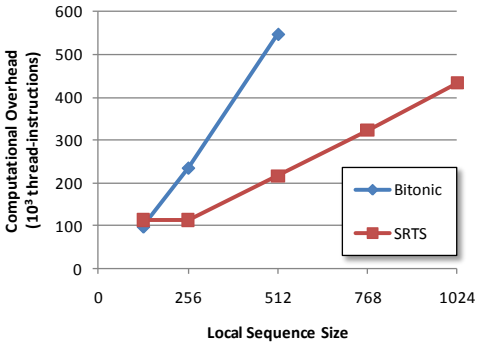

Figure 18. Local workloads (excluding I/O instructions) for local bitonic and SRTS radix sorting networks.

The overall memory workloads for these implementations are shown in Figure 15. We confirm our model from Section 4.1 that memory overhead monotonically decreases with increasing *d*. The memory workload of the CUDPP implementation (4-bit digits) is 1.6x that of our 4-bit variant.

Figure 16 illustrates the memory overheads for a single distribution sort, broken down by kernel and type. Although our scatter instructions logically write two 32-bit words per pair (the key and the value), we observe that the hardware issues additional write transactions when threads within the same half-warp write keys having different radix digits to different memory segments. On this architecture, these write instructions incur a ~70% I/O overhead that increases in proportion with *r*, the number of digit partitions. This non-coherence overhead accounts for a 28% increase over our cumulative I/O model; the CUDPP implementation experiences a 22% increase over its cumulative I/O model.

The scatter inefficiencies decrease for less-random key distributions. With zero effective random bits (uniformly identical keys), our 4-bit implementation averages a saturated sorting rate of 550 x10^6 pairs/second. These compute-bound kernels do not benefit from this lower memory workload: this speedup is instead gained from the elimination of warp-serialization hazards that stem from bank conflicts incurred during key exchange.

The boundary between compute-boundedness and memory-boundedness, or the *memory wall*, is the ratio between computational and memory throughputs. It is often expressed in terms of the number of cycles that can execute per memory-reference, but we prefer the inverse: the average number of bytes that can be serviced per cycle. Figure 17 illustrates the corresponding workload ratios for each of the stream kernels relative to the GTX-285 memory wall ($\delta_{mem}/\delta_{compute} \approx 4.5$ bytes/cycle). We see that for $d > 2$, our distribution sorting streams do not oscillate between and memory-bound and

compute-bound kernels. The CUDPP implementation contains a mixture of memory-bound and extremely compute-bound kernels.

For input sequences smaller than 2,048 keys, it is more efficient to implement the entire set of k/d distribution-sorting passes as a local sorting problem, i.e., within a single kernel consisting of a single CTA. Keys are read and written only once from global memory and exchanged k/d times in local shared memory. Bitonic sorting networks [39] are often used for such local sorting problems; Figure 18 compares the computational overhead of our local radix sort versus that of the bitonic sorting implementation provided by the NVIDIA CUDA SDK [40]. The efficiency of our radix sort exceeds that of bitonic sort for sequences larger than 128 keys, exhibiting a speedup of 2.5x for 512 keys. (Our 4-bit implementation incurs a constant overhead for n smaller than the 256-element local block-exchange size; the bitonic implementation is limited to $n \leq 512$, the maximum number of threads per CTA.)

6 Discussion of Related Work and Conclusion

We have presented efficient radix-sorting strategies for ordering large sequences of fixed-length keys (and values) on GPUPU stream processors. Our empirical results demonstrate multiple factors of speedup over existing GPGPU implementations, and we believe our implementations to be the fastest available for any fully-programmable microarchitecture. These results motivate a different breed of parallel primitives for GPGPU stream architectures that can better exploit the memory and computational resources while maintaining the flexibility of a reusable component. Our generalized parallel scan stream primitive does this in two ways: (1) with interfaces for producer/consumer operations (*visiting logic*) in order to increase the arithmetic intensity of the memory-bound parallel prefix scan primitive in an application-neutral manner; and (2) with interfaces for performing multiple related, concurrent prefix scans (*multi-scan*) in order to increase the arithmetic intensity in an application-specific way by increasing the size of the radix digits.

Although they can be very efficient, radix sorting methods make certain positional and symbolic assumptions regarding the bitwise representations of keys. A comparison-based sorting method is required for a set of ordering rules in which these assumptions do not hold. A variety of comparison-based, top-down partitioning and bottom-up merging strategies have been implemented for the GPGPU, e.g., quicksort [41,15], most-significant-digit radix sort [32], sample-sort [42,43], and merge sort [19]. The number of recursive iterations for these methods is logarithmic in the size of the input sequence, typically with the first or last 8-10 iterations being replaced by a small local sort within each CTA.

There are several contributing factors that give radix sorting methods an advantage over their comparison-based counterparts. First, comparison-based sorting methods must have work-complexity $O(n\log_2 n)$ [2], making them less efficient as problem size grows. Second, for problem sizes large enough to saturate the device (e.g., several hundred-thousand or more keys), a radix digit size $d \geq 4$ will result in fewer digit passes than recursive iterations needed by the comparison-based methods performing binary partitioning. Third, the amount of global intermediate state needed by these methods for a given level in the tree of computation is proportional to the width of that level, as opposed to a small constant amount for our radix sort strategy. Finally, parallel radix sorting methods guarantee near-perfect load-balancing amongst GPGPU cores, an issue of concern for comparison-based methods involving pivot selection.

We have also demonstrated a method for encoding multiple binary-valued flag vectors into a single, composite representation. This allows us to execute several compaction tasks in shared memory while only incurring the cost of a single parallel scan. While this technique allows us to increase the number of digit bits d more than we would be able to otherwise, it is not a critical ingredient for our speedup. Without flag vector encoding, our $d = 2$ bits distribution sort would require four local scans, the same computational overhead as our 4-bit distribution sort. This four-scan distribution sort exhibits a sorting rate of 3.9 $\times 10^9$ pairs/second; sixteen passes would result in an overall sorting rate of 241 $\times 10^6$ pairs/second (and a speedup of 1.8x over the CUDPP implementation).

For small sorting problems suitable for a single GPU core, this technique allows our local sorting implementations to be several times more efficient than popular bitonic and odd-even sorting networks. Bitonic networks have proven convenient

for mapping onto GPGPU computation [44,15,16,45,46]. Although their $O(n\log_2^2 n)$ work complexity scales poorly for large problems, their simplicity has made them amenable for sorting small, local sequences of keys [19,15]. Our work provides an attractive, drop-in replacement for stream applications that currently utilize local bitonic sorting methods.

7 References

[1] J D Owens et al., "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879-899, May 2008.

[2] Donald Knuth, *The Art of Computer Programming*. Reading, MA, USA: Addison-Wesley, 1973, vol. III: Sorting and Searching.

[3] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein, *Introduction to Algorithms*, 2nd ed.: McGraw-Hill, 2001.

[4] Erik Sintorn and Ulf Assarsson, "Real-time approximate sorting for self shadowing and transparency in hair rendering," in *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, Redwood City, California, 2008, pp. 157--162.

[5] Kun Zhou et al., "RenderAnts: interactive Reyes rendering on GPUs," in *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers*, Yokohama, Japan, 2009, pp. 1--11.

[6] Bernhard Kainz et al., "Ray casting of multiple volumetric datasets with polyhedral boundaries on manycore GPUs," in *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers*, Yokohama, Japan, 2009, pp. 1--9.

[7] Peter Kipfer, Mark Segal, and Rüdiger Westermann, "UberFlow: a GPU-based particle engine," in *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Grenoble, France, 2004, pp. 115--122.

[8] Jonathan M Cohen, Sarah Tariq, and Simon Green, "Interactive fluid-particle simulation using translating Eulerian grids," in *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, Washington, D.C., 2010, pp. 15--22.

[9] Charles Loop and Kirill Garanzha, "Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing," in *Eurographics*, 2010.

[10] Ignacio Castaño. (2007, February) High Quality DXT Compression Using CUDA. [Online]. http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/dxtc/doc/cuda_dxtc.pdf

[11] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo, "Real-time KD-tree construction on graphics hardware," in *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*, Singapore, 2008, pp. 1-11.

[12] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha, "Fast BVH Construction on GPUs.," *Comput. Graph. Forum*, vol. 28, pp. 375-384, 2009.

[13] B Fabianowski and J Dingliana, "Interactive Global Photon Mapping," *Computer Graphics Forum*, vol. 28, pp. 1151-1159(9), June-July 2009.

[14] Dan A Alcantara et al., "Real-time parallel hashing on the GPU," in *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers*, Yokohama, Japan, 2009, pp. 1--9.

[15] Bingsheng He et al., "Relational joins on graphics processors," in *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, Vancouver, Canada, 2008, pp. 511--524.

[16] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha, "GPUTeraSort: high performance graphics co-processor sorting for large database management," in *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, Chicago, IL, 2006, pp. 325--336.

[17] Naga Govindaraju, Nikunj Raghuvanshi, and Dinesh Manocha, "Fast and approximate stream mining of quantiles and frequencies using graphics processors," in *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, Baltimore, MD, 2005, pp. 611--622.

[18] Jeremy Shopf, Joshua Barczak, Christopher Oat, and Natalya Tatarchuk, "March of the Froblins: simulation and rendering massive crowds of intelligent and detailed creatures on GPU," in *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, Los Angeles, CA, 2008, pp. 52--101.

[19] Nadathur Satish, Mark Harris, and Michael Garland, "Designing efficient sorting algorithms for manycore GPUs," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, 2009, pp. 1-10.

[20] GPGPU.org. [Online]. http://gpgpu.org/developer/cudpp

[21] Jatin Chhugani et al., "Efficient implementation of sorting on multi-core SIMD CPU architecture," *Proc. VLDB Endow.*, pp. 1313-1324, 2008.

[22] Larry Seiler et al., "Larrabee: a many-core x86 architecture for visual computing," in *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, Los Angeles, CA, 2008, pp. 1-15.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addisson-Wesley, 1995.

[24] Duane Merrill and Andrew Grimshaw, "Parallel Scan for Stream Architectures," University of Virginia, Department of Computer Science, Charlottesville, VA, USA, Technical Report CS2009-14, 2009.

[25] Leslie G Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, pp. 103-111, 1990.

[26] A Hartstein, V Srinivasan, T R Puzak, and P G Emma, "Cache miss behavior: is it √2?," in *CF '06: Proceedings of the 3rd conference on Computing frontiers*, Ischia, Italy, 2006, pp. 313-320.

[27] Guy Blelloch, "Prefix Sums and Their Applications," School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-90-190, 1990.

[28] Siddhartha Chatterjee, Guy Blelloch, and Marco Zagha, "Scan primitives for vector computers," in *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, New York, New York, 1990, pp. 666-675.

[29] Andrea C Dusseau, David E Culler, Klaus E Schauser, and Richard P Martin, "Fast Parallel Sorting Under LogP: Experience with the CM-5," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, pp. 791-805, 1996.

[30] Marco Zagha and Guy Blelloch, "Radix sort for vector multiprocessors," in *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Albuquerque, NM, 1991, pp. 712--721.

[31] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D Owens, "Scan Primitives for GPU Computing," in *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, San Diego, CA, 2007, pp. 97--106.

[32] Bingsheng He, Naga K Govindaraju, Qiong Luo, and Burton Smith, "Efficient gather and scatter operations on graphics processors," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, Reno, NV, 2007, pp. 1-12.

[33] Guy E Blelloch, Siddhartha Chatterjee, and Marco Zagha, "Solving Linear Recurrences with Loop Raking," in *Proceedings of the 6th International Parallel Processing Symposium*, 1992, pp. 416-424.

[34] Mark Harris. (2007) Optimizing parallel reduction in CUDA. [Online]. http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf

[35] Yuri Dotsenko, Naga K Govindaraju, Peter-Pike Sloan, Charles Boyd, and John Manferdelli, "Fast scan algorithms on graphics processors," in *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, Island of Kos, Greece, 2008, pp. 205-213.

[36] Peter M Kogge and Harold S Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *IEEE Trans. Comput.*, vol. 22, pp. 786-793, 1973.

[37] Shubhabrata Sengupta, Mark Harris, and Michael Garland, "Efficient Parallel Scan Algorithms for GPUs," NVIDIA, Technical Report NVR-2008-003, 2008.

[38] Markus Billeter, Ola Olsson, and Ulf Assarsson, "Efficient stream compaction on wide SIMD many-core architectures," in *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, New Orleans, LA, 2009, pp. 159-166.

[39] K E Batcher, "Sorting networks and their applications," in *AFIPS '68 (Spring): Proceedings of the April 30--May 2, 1968, spring joint computer conference*, Atlantic City, NJ, 1968, pp. 307-314.

[40] NVIDIA. (2010, February) NVIDIA CUDA SDK - CUDA Basic Topics. [Online]. http://www.nvidia.com/content/cudazone/cuda_sdk/CUDA_Basic_Topics.html

[41] Daniel Cederman and Philippas Tsigas, "GPU-Quicksort: A practical Quicksort algorithm for graphics processors," *J. Exp. Algorithmics*, vol. 14, pp. 1.4--1.24, 2009.

[42] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders, GPU sample sort, 2009.

[43] Frank Dehne and Hamidreza Zaboli, Deterministic Sample Sort For GPUs, 2010.

[44] Peter Kipfer and Rüdiger Westermann, "Improved GPU sorting," in *GPUGems 2*, Matt Pharr, Ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2005, ch. 46, pp. 733-746.

[45] A Greb and G Zachmann, "GPU-ABiSort: optimal parallel sorting on stream architectures," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, 2006, p. 10 pp.

[46] Mark Harris, Shubhabrata Sengupta, and John Owens, "Parallel Prefix Sum (Scan) with CUDA," in *GPU Gems 3*, Hubert Nguyen, Ed. Boston, MA: Addison-Wesley, 2007, ch. 39.